計算機アーキテクチャ 79－1
（１９８９．１１．２０）

マルチプロセッサ上の同期機構と プロセッサスケジューリングに関する考察

松 本 尚

日本アイ・ビー・エム株式会社　　東京基礎研究所

マルチプロセッサシステムにおいて，複数のプロセッサに協調して処理を行わせるにはプロセッサ間の同期が不可欠である。共有メモリ型のマルチプロセッサシステム特にスヌープ・キャッシュ付きのシステムでは，比較的オーバーヘッドの少ない同期法としてプロセッサ間で共有した同期変数を用いたビジー・ウェイトが利用される。しかし，オペレーティングシステムの下ではユーザーは実プロセッサを仮想化した枠組みプロセス(やスレッド)を用いて処理を記述しているため，同期相手のプロセスがプロセッサにスケジューリングされておらず，同期がすぐには成立せずプロセッサが無駄なビジー・ウェイトを行うといった事態が起こる。本稿ではこの問題を踏まえて，プロセッサ資源の浪費を抑えたオーバーヘッドの少ない同期を可能にするために以下の四つの機構を提案する。(1) 実プロセッサ資源のプロセスへの割当状況をビジー・ウェイトのループ内でチェックする新しいビジー・ウェイト同期法。同期が当分成立しないと判断される場合はビジー・ウェイトを行っているプロセスは自ら実行権を放棄する。(2) この新しい同期法のアルゴリズムの一部をハードウェア化した同期機構。新たに追加されたチェックのためのオーバーヘッドを減らす。(3) ユーザーのスケジューラとカーネルのスケジューラの二段構造になった階層化されたスケジューラ。スケジューリング法のユーザーによるカスタマイズとカーネルへの切り替わりのオーバーヘッドの除去を可能にする。(4) ユーザー・モード内での割り込みとカーネル・モードへの割り込みをサポートするプロセッサの割り込みシステム。プロセス切り替えに関する同期機構からの割り込みのオーバーヘッドを緩和する。

Synchronization and Processor Scheduling Mechanisms for Multiprocessors

Takashi MATSUMOTO

IBM Research, Tokyo Research Laboratory
5-19, Sanbancho, Chiyoda-ku, Tokyo 102

In a multiprocessor system it is necessary to synchronize cooperating processors. Busy-waiting on a shared variable is commonly used in shared memory multiprocessors, especially those with snoopy cache. However, busy-waiting may be inefficient when programs are run under an operating system, where programs are written in terms of processes (or threads) which are abstraction of processors: a process may busy loop in vain waiting for another process to be scheduled to a processor. This paper proposes the following mechanisms for efficient synchronization and processor scheduling: (1) A new busy-waiting mechanism. Allocation of processes to processors is monitored within a busy loop, and if it is detected that the synchronization will not be established for some time, the waiting process yields execution to another process. (2) A hardware support mechanism for this busy-waiting mechanism, which reduces the cost introduced by the additional checking. (3) A scheduler divided into the user and the kernel levels, which realizes policy-mechanism separation and avoids the overhead of crossing the user-kernel boundary. (4) An interrupt system for handling interrupts at user-mode and at kernel-mode, which supports interrupts specifying process switching in the synchronization mechanism.

1.はじめに

VLSI技術の大きな進歩により，多数のプロセッサを搭載したマルチプロセッサシステムが実現されるようになった。そして，複数のプロセッサを協調動作させて仕事を高速に処理する並列処理が重要になってきている。複数のプロセッサを協調させて処理を行わせるにはプロセッサ間の同期が不可欠である。データの依存関係に基づく同期に関しては，データ・フロー・マシンやメッセージ・パッシング型のマシンでは同期はデータの通信と不可分で，データの通信と同時に同期も取られる。しかし，最近増えてきた共有メモリ型のマルチプロセッサシステムで共有メモリを介してのデータの通信を行うと，プロセッサは他のプロセッサからのデータ通信の完了または未完を自然に知ることはできない。このため，データの通信と別個に同期のための処理を行う必要がある。また，クリティカル領域における排他制御のためにも同期は不可欠である。本稿では共有メモリ型のマルチプロセッサシステムの同期問題について論じる。

マルチ・ジョブやマルチ・ユーザーの機能の実現のために，現実のハードウェア資源の管理はオペレーティングシステム（以下OS）によって行われていると仮定する。そして，ユーザーやプログラマは実プロセッサを仮想化した枠組みを用いて処理を記述する。ここではこれをプロセスと呼ぶ。プロセスは OS の管理の下で実プロセッサの割当てを受け処理を実行する。この割当てをプロセススケジューリング（以下単にスケジューリング）と呼び，一つのプロセスに対し一つの実プロセッサが割当てられるとする。また，プロセッサの割当を行う部分をスケジューラと呼ぶ。

OSの下の並列処理では協調して動作する複数のプロセスを用意して，プロセス間で同期を取りながら処理を進める。この同期の方法には一般には次の二つの方法が用いられている。一つは OS を介して同期を行う方法で，もう一つはプロセス間の共有メモリを介して同期を行う方法である。 OS を介して同期を実行する場合は，同期が成立しなかったプロセスのプロセッサの割当てを外し休止状態にして，空いたプロセッサには他のプロセスを割付けることが可能である。これにより，プロセッサ資源を効率的に使うことができる。しかし，OS を介しての同期は OS（カーネル）とユーザーの間の制御権の切り替え時のオーバーヘッドが大きく，休止状態になり再びプロセッサに割付けられるといったことを繰返すと，さらにオーバーヘッドが増大する。オーバーヘッドを少なくするため OS を介しての同期ではなく共有メモリを介してのビジー・ウェイトが共有メモリ型のマルチプロセッサシステムではよく行われる。しかし，ビジー・ウェイトで同期を行うとき，逆に問題が大きくなる場合がある。スケジューリング時に，複数の特定のプロセスを同時に複数のプロセッサに割付るといった指定が通常できない。また，それが可能なシステムでも，実プロセッサの台数を越えたプロセスを同時に割り付けることはできない。スケジューリングの都合で，同期を取り合うプロセス群の内のいくつかがプロセッサの割当て待ちの状態になっているとすると，同期時に相手のプロセスがプロセッサにスケジューリングされておらず，同期がすぐには成立せずプロセッサが無駄なビジー・ウェイトを行うといった事態が起こる。この場合，CPU時間は消費するが，タイム・スライスか何かで再スケジューリングが起こって，相手のプロセスが実プロセッサに割当てられるまで，プログラムは進展しない。

これまでの議論でも判るように，プロセスの同期とスケジューリングは密接な関係を持っている。アプリケーションや特定の並列処理プログラムにおいては，スケジューリングを調節することによって効率を上げることができる。しかし，現状のOS では総てのプロセスを同じアルゴリズムに基づいてスケジューリングを行っているのでアプリケーション毎のスケジューリングの調節といったことが自由にはできない。また，並列性の大きな問題を自然にプログラミングして多数の細かい粒度のプロセスを生成する場合，頻繁にスケジューラが起動する。こういった処理も効率よく実行できるためにスケジューラのオーバーヘッドを小さくする必要がある。

以上に述べてきた問題点を解決するために，以下で述べる方式並びに機構を提案する。2章ではスケジューリング不具合によるプロセッサの浪費を抑える新しいビジー・ウェイト同期方式，3章ではその一部をハードウェア化してオーバーヘッドを減らした同期機構， 4章ではユーザーによるスケジューリング法のカスタマイズを可能にする軽い階層化されたスケジューラ， 5章ではハードウェア化された同期機構と組み合わせてオーバーヘッドをさらに減じるプロセス間の非同期通信のための軽い割り込み機構について述べ，6章でまとめを述べる。

なお，ハードウェア化する部分については，筆者らが提唱している FGSM (Fine Grain Support Multiprocessor)[1]に簡単に組み込み可能なように一般化されたバリア型同期機構[2] [3]となるべく同じ構成を取るように考慮されている。

2. 新しいビジー・ウェイト同期方式

2.1. プロセッサ資源の浪費の解決法

同期を軽くすることを目的の一つとしているので，共有メモリ等を介してのビジー・ウェイトで同期を行うことを前提にする。また，ビジー・ウェイト中の共有メモリへのアクセスの集中がデータ通信路のトラフィックを増大させてシステムの性能を低下させないため，コンシステンシを保つキャッシュ（スヌープ・キャッシュ[4]等）がプロセッサ毎に用意されていることを仮定する。

1章でも述べたように，「無駄に同期待ちを行うプロセスがプロセッサに割り付けられて，プロセッサ資源を浪費する。」という問題が存在する。ここで，「無駄に」と表現しているのは実プロセッサの割当てを待っているプロセスに対する同期をビジー・ウェイトのループで待つことである。スケジューラが起動して再スケジューリングされて，その対象のプロセスがプロセッサに割当てられるまで，無駄にループを回り続ける。このプロセッサ資源の浪費を回避するためには，プロセスが「無駄に」同期待ちを行っていることが認識できなければならない。そこで，ビジー・ウェイトの同期待ちのループでは同期変数のチェックだけではなく，システムのプロセッサ資源に関する情報つまりプロセスの実プロセッサへの割当状況もチェックする。そのチェックの結果，状況によってはプロセスを中断し，スケジューラに制御を移し，再スケジューリングを実行しプ

ロセッサへのプロセスの割り当てを変更する。この方式により無駄に同期待ちを行うプロセッサを減らすことができる。

2.2. プロセス切り換えの状況判断

プロセスが同期待ち時にプロセッサの割当てを自ら放棄する条件について述べる。ここで，プロセスとプロセッサ資源を有効に管理するためにグループという考え方を導入する。基本的にはプロセスのうち共有メモリを介した同期を行うもの同士を一つのグループとする。つまり，グループが異なれば，共有メモリを介したビジー・ウェイトの同期を行わない。プロセッサは自分の上で現在実行されているプロセスのグループに属する。同じグループのプロセスが同時に割り付けられているプロセッサ群がその時点でのプロセッサのグループを形成する。

同期待ち時に，実行中のプロセスが自ら実行を中断し，再スケジューリングを要求すべき条件の例として以下のようなものが挙げられる。

(1) 同期を成立させる相手のプロセスが判っていて，そのプロセスがプロセッサ割当て待ちである時。

(2) クリティカル領域への排他制御で，領域内に入る権利を持っているプロセスがプロセッサ割当て待ちで寝ている時。

(3) 自分と同じグループに属するプロセッサが総て同期待ちになり，プロセッサ割当て待ちのプロセスが存在する時。

(4) 自分と同じグループに属するプロセスが総て同時にプロセッサに割り付けられており，それら総てが同期待ちになった時。但し，これはプログラミング・エラー（デッド・ロック）と考えられる。

(5) バリア同期を行うプロセス群で一つのグループを形成しており，自分がそのグループの一員の場合，自分のグループに属するプロセッサ割当て待ちかつ同期待ちでないプロセスが存在する時。

(6) 自分と同じグループに属する同期待ちのプロセッサの数が nを越え，プロセッサ割当て待ちのプロセスが存在する時。但し，nは OSまたはユーザーによって設定された値。

(7) 自分と同じグループに属する同期待ちのプロセッサの数が nを越え，プロセッサ割当て待ちの自分と 同じグループのプロセスが存在する時。

(1) (2) (3) (4) (5) の条件はプロセスを交代する方が確実に効率が向上する条件で，(6) (7) の条件は発見法的に経験則から nの値を決めて効率の向上を狙うものである。条件(1) (2) はきめ細かく制御できるが，逆に同期毎にその同期を成立させるプロセス等を細かく指定する必要がある。これに対し，(3) (4) (5) (6) (7) の条件は指定がグループ・レベルなので，同期毎の異なった指定は必要ない。(6) (7) の条件については，処理しているアプリケーションによって nの値を調節して効率を上げるべきである。また，同期待ちの絶対数で判断するのではなく，同じグループに属するプロセッサ数とその内の同期待ちの数の比率を基準にしてもよい。

2.3. 状況判断に必要な情報

これらの条件を満たすかどうかチェックするのに必要なシステムのプロセスとプロセッサ資源に関する情報は，基本的には，プロセスの実プロセッサへの割当て状況とプロセスのグループ形成状況である。

条件(1) では同期毎に同期を成立させる相手のプロセスを知っていることが前提になっているので，そのプロセスがプロセッサに割り付けられているかどうか低コストで調べることができれば良い。そこで，例えば，グループ内のプロセス識別番号とシステムのプロセス識別番号の対応表とプロセス識別番号とプロセッサ割当ての対応表といったものがユーザーからアクセスできる共有メモリ上に用意されていればよい。

条件(2) ではクリティカル領域に入る権利（鍵）を現在持っているプロセスを識別する必要がある。そこで，権利を獲得すると必ず自分のプロセス識別番号をクリティカル領域毎に決められた場所に格納することにし，権利を放棄する際に初期化することにする。当然，排他制御の鍵や識別番号の格納場所はグループ内のプロセスからユーザー・モードで読み書き可能な共有メモリ上に用意される。

条件(3)〜(7) では基本的に，同一グループの同期待ちのプロセッサ数が一定数に達した時に条件が成立している。今後，グループに係わるプロセス数とプロセッサ数の情報を以下の記法で記述することにする。

- 自分のグループに属するプロセッサの数，（グループに属するプロセスの内でプロセッサに割り付けられているプロセスの数）記号: #MGC (the number of My Group Cpus)

- 自分のグループに属するプロセッサの内，同期待ちのプロセッサの数，記号: #MWC (the number of My group Waiting Cpus)

- プロセッサ割当て待ちのプロセスの数，記号: #PRQ (the number of Processes in Run Queue)

- 自分のグループに属するプロセッサ割当て待ちのプロセスの数，記号: #MPRQ (the number of My group Processes in Run Queue)

- 自分のグループに属するプロセッサ割当て待ちだが同期待ちでないプロセスの数，記号: #MNWR (the number of My group Not Waiting processes in Run queue)

この記法を使って条件(3)〜(7)を書き直すと，以下のようになる。

(3) $(\#MWC \geq \#MGC)$ and $(\#MPRQ > 0)$
(4) $(\#MWC \geq \#MGC)$ and $(\#MPRQ = 0)$

(5) ($\#MWC \geq 1$) and ($\#MNWR > 0$) at Barrier Sync.
(6) ($\#MWC \geq n$) and ($\#PRQ > 0$)
(7) ($\#MWC \geq n$) and ($\#MPRQ > 0$)

これらの情報がユーザーのプロセスから低コストで参照できる必要があり，共有メモリ上にカーネル（OS）側からもユーザー側からもアクセスできる変数として置かれる。アクセス権の保護に関しては，#MWC と #MNWR のみ同期を行うユーザーのアプリケーションのプロセスからも書き込み可能で，他はスケジューラ側のみから書き込める。スケジューラはこれらの値をスケジューリング毎に必要に応じて更新する。

2.4. 新しいビジー・ウェイト・ループ

プロセス切り換えのための条件判断を含む効率のよいビジー・ウェイト同期方式についてフローチャートに沿って説明する。同期変数のチェックやプロセス切り換えの条件判断の具体的な方法等は，待ち合わせを行うプロセスの数や同期の種類によって異なる。しかし便宜上，ここでは図1 の一つのフローチャートで待ち合わせ処理の内容を示し，細かい差は無視する。また，プロセス切替条件(1)(2)(5) を調べるだけならば，#MWC の更新の必要はない。同期待ち状態突入処理と脱出処理はスケジューラに対してスケジューリング時に利用できる情報を伝えるための処理である。例えば，「このプロセスは同期待ち中である。」とか「プロセス NO.123 からの同期の承認を待ってる。」とか「同期識別番号 NO.9876 の同期を取っている。」といった情報をメモリに書き込み，スケジューリング時にスケジューラに利用してもらって，効率の良いスケジューリングを行ってもらう。

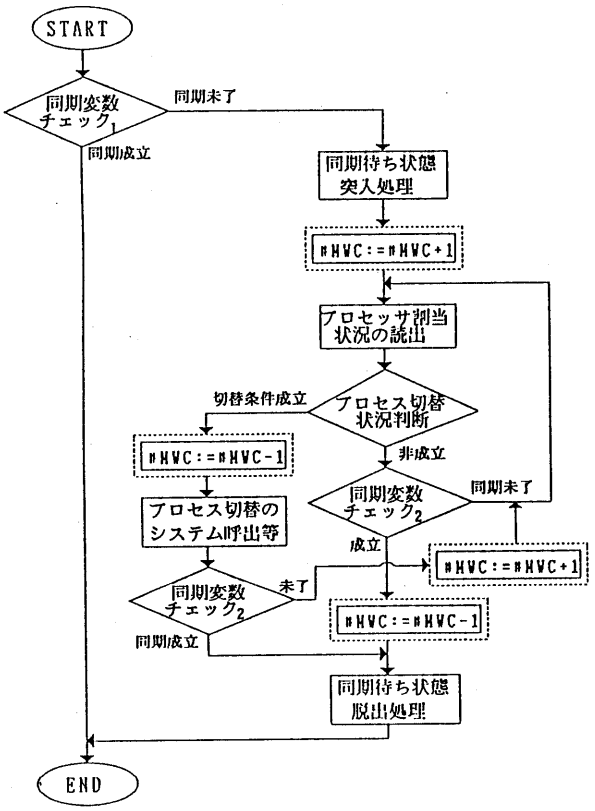

図1　新しいビジー・ウェイト・ループ

図2 は従来の待ち合わせのためのループを示す。最良の条件，つまり同期変数を初めてチェックするまえに同期が成立し同期変数に同期成立の値がセットされている場合において，図1 の方式のオーバーヘッドを従来の方式と同じに抑えるために処理の頭で同期変数のチェックを一度行っている。同期完了であれば当然すぐに待ち合わせ処理を終了する。一回目の同期変数のチェックで同期未了の場合に限り，プロセッサが同期待ち状態に入り，これにより影響を受ける変数（#MWC 等）を更新する。そして，システムのプロセッサ割当て状況に関する情報を読出して，前出の条件によってプロセスを中断しスケジューラによるプロセスの再スケジューリングを要求するかどうか判断する。条件が成立すれば，プロセッサの制御権を放棄するシステム呼出し等を行ってスケジューラを呼び出す。条件が成立していなければ，新たに同期変数のチェックを行なう。同期が成立していなければ，プロセッサ割当て状況に関する情報の読出しに戻って繰返す。同期が成立していれば，プロセッサが同期待ち状態を抜け，これにより影響を受ける変数（#MWC 等）を更新し，待ち合わせの処理を終了する。制御権を放棄して，再びプロセッサに割り付けられた時は，プロセス切り換えの条件判断が非成立の場合のフローに合流する。

図1で同期変数のチェックが二種類存在する。チェック1 の方は従来の物と全く同じである。チェック2 の方は同期の完了のチェックと同期完了後の同期変数の後始末のみを行っている。例えば，カウンタを用いたバリア同期ではバリアに達したことを示すカウンタの増減の処理はチェック1 にだけ含まれる。

図1で点線で囲まれた処理の部分は共有メモリへのアクセスを不可分で排他的に行うべき部分（#MWCの更新）を示す。つまり，ロックを掛けてアクセスする部分である。

システムがスヌープ・キャッシュ等を持つことを前提としているので，図3 のようにスピン・ロックを行えばオーバーヘッドを増やさずに無駄なプロセス切り換えの要求を抑えられる。つまり，図1 の方式では複数のプロセスで同時にプロセス切り換えの条件が成立し，プロセス切り換え要求が集中する可能性がある。そこで，図3 のように実際にプロセス切り換えを要求する部分をクリティカル領域と

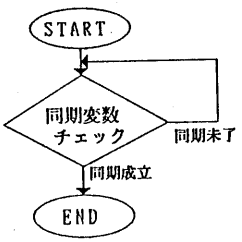

図2　従来のビジー・ウェイト方式

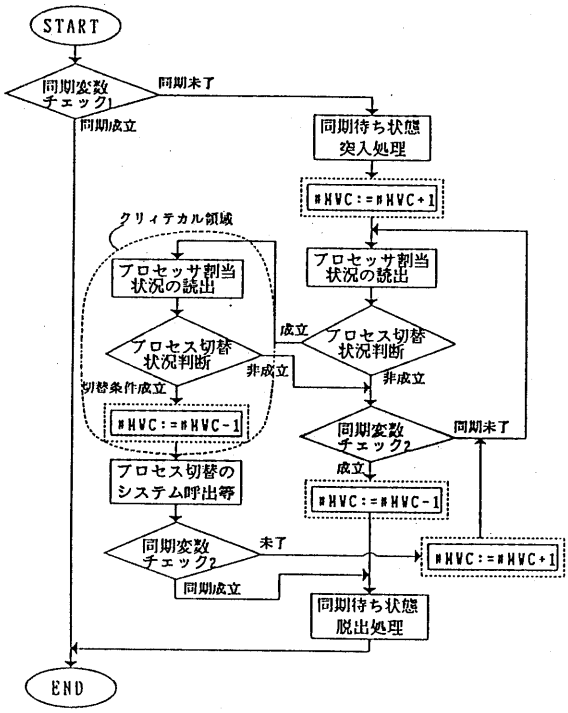

図3　プロセス切替の集中を緩和した
新しいビジー・ウェイト・ループ

して排他制御することにより，その要求の集中を防いでいる。しかし，ビジー・ウェイトの最内側ループにクリティカル領域の排他制御のための共有メモリへの排他アクセス（バス・ロック等）が入ると，共有メモリへのアクセスが集中してオーバーヘッドが大きくなる。そこで，最内側ループではクリティカル領域とせずにプロセス切り換えの条件判断を行ない，条件が成立したときに限り，クリティカル領域に入り条件判断をし直す（スピン・ロック）。なお，図3 の方式の場合，スケジューラがプロセスを切り換えてシステムのプロセッサ資源に関する変数を更新する際も，アクセスは排他制御される。

図1 図3 には省略されているがバリア同期で条件(5) を用いる場合は同期が成立した時点で #MNWR を更新する必要がある。この際，最初に同期成立を確認したプロセスが他のバリア同期を行っている同じグループのプロセッサがすべて同期の成立を認識したことを確認後 #MNWR を #MPRQ に再初期化する。つまり，バリア同期待ちでかつプロセッサの割当て待ちだったプロセスは同期の成立と共にバリア同期待ちではなくなるので，この変化を反映させる。

3. 新しい同期方式を支援する機構

3.1. 機構の基本方針

粒度が比較的小さいプログラムでは新しい方式のオーバーヘッドが従来方式に比べて目立ってくる。同期待ち状態でのループの繰返し毎の処理量は図2 と図3 から比べても判るように，新しい方式は従来に比べてかなり重たい。そのため，ループを 1〜2回しか回らない場合は，同期が成立した（同期変数を誰かが書き替えた）時点から実際に成立を検出して待ち合わせ処理を終了するまでのオーバーヘッドが目立つようになる。また，#MWC の更新が排他的な共有メモリへのアクセスでコストが掛かるので，これを各プロセッサに局所的な処理で済ませたい。

プロセス切替の条件(1)(2) に関しては，同期毎に必要な処理が変化するのでハードウェア化には向いていない。この章では，残りの五つの条件(3)〜(7)のチェックをハードウェアの支援で軽くすることを考える。 2.3.節でも述べたように，これらの条件では基本的に #MWC が一定数に達したことをチェックしている。そこで，プロセッサ外部からハードウェア的にプロセッサが同期待ち状態にいるかどうか検知できる機構を用意して，同期待ち状態のプロセスを切り換えるべきかどうかのチェックを専用ハードウェアに行わせ，割り込みで条件成立を通知すれば，プロセッサは図4 のような待ち合わせ処理で済む。これにより前出の場合でも従来方式と遜色がなくなる。なお，ステラのマシンも専用レジスタで同期を取る場合，条件(3)(4) の成立を割り込みで通知する機構を持っている[5]。よって，本稿で述べる機構はこれをプロセス切替の条件の追加や同期変数の共有メモリ上への割当ての自由化等で拡張したものになっている。

また，本稿の機構はバリア同期における排他的メモリ・アクセスの集中の問題も同時に解決する。つまり，一つの同期変数を用いたN台のプロセッサによるバリア同期の場合，少なくともN回の同期変数への排他的な操作が必要である。この部分のメモリ処理は直列化されるのでNが大きくなるとオーバーヘッドも増大する。特に，バリア同期を取るプロセス間の粒度が揃っており，それらのプロセスが総て同時にプロセッサに割り当てられている状況で，このオーバーヘッドは問題となる。このオーバーヘッドを避けるためにバリア同期専用の同期変数を専用レジスタ（本機

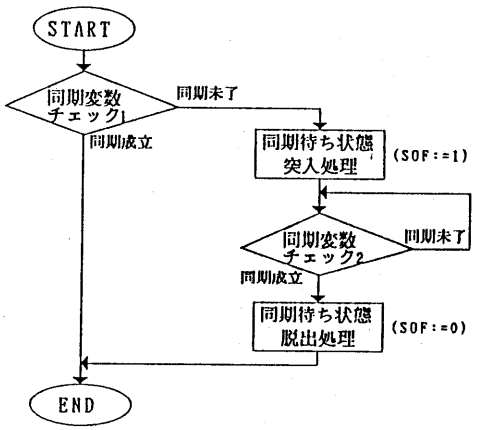

図4　同期支援ハードウェアがある場合の
ビジー・ウェイト・ループ

構ではフラグ）としてプロセッサ毎に設け，レジスタ間の値の更新は同期情報のための専用通信路（本機構ではブロード・キャスト型の信号線）で行うことにする。

この専用レジスタと専用通信路を使って同期のオーバーヘッドを軽減する機構は多くのシステムで採用されている[6][7]。他の機構との差はバリア型の同期であることを生かして，同期を行うプロセスが同時にプロセッサに割り当てられなくてもプロセッサを最大限に効率的に利用して処理を進めることができる点である。また，一般化されたバリア型同期機構のように，専用レジスタを命令でチェックする代わりにハードウェア的にプロセッサを休止状態にして同期を取る方法も考えられる。しかし，念頭にある粒度が本機構の方が少し大きいことと使用するプロセッサに制約がないことから，専用レジスタをチェックしながらビジー・ウェイトする方式を採用する。

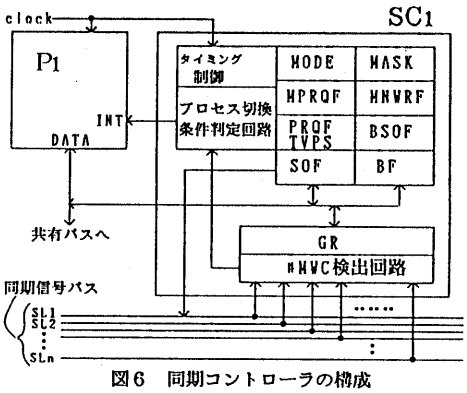

図6　同期コントローラの構成

3.2. 機構の構成例

同期機構の構成例について述べる。図5 に全体構成を示す。データ通信路は共有バスとした。プロセッサ毎に同期コントローラが設けられ，同期コントローラ間はプロセッサ台数分の信号線を持つ同期信号バスで結合されている。各プロセッサと同期コントローラの間はコントローラ内のレジスタやフラグを読み書きするためのデータ線とコントローラからプロセッサへの割り込み線で結合している。図6　に同期コントローラの構成を示す。同期コントローラ毎に同期信号バス上の特定の一本の信号線が割り当てられており，その信号線に対してのみ信号（'0'か'1'の2値）を出力できる。この出力はコントローラ内の同期待ち出力フラグ(SOF)　に対応しており，SOFがセットされると信号線に'1'が出力され，SOF がリセットされると '0' が出力される。初期状態として SOF はリセットされており，プロセッサが共有メモリを介しての同期待ちのループに入るまえに SOF をセットし，ループを抜けるとリセットするようにプログラミングする（図4 参照）。これにより，同期待ち状態のプロセッサに対応する同期信号バスの信号線は総て'1'になる。また，同期コントローラ内には各同期信号線に各ビットが対応するグループ・レジスタ(GR) があり，自分と同じグループに属するコントローラの対応するビットに'1'がスケジューラによってセットされている。これにより，同期コントローラは自分の属するグループのプロセッサが同期待ち状態にあるかどうか区別できる。本同期コントローラは 2つの動作モードを持って

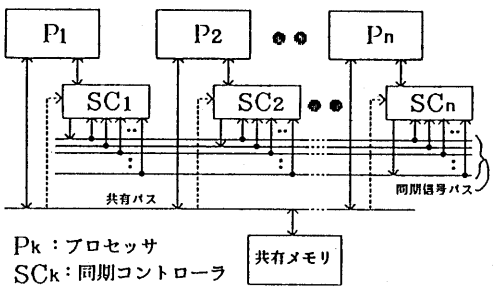

図5　システムの全体構成

おり，モード1 は共有メモリを介したビジー・ウェイトに対応し，モード2 は専用フラグを用いたバリア同期に対応している。同じグループ内では一時点にはどちらか一方のモードしか選択できないが，グループが異なればモードは異なっても構わない。モードの切り換えはコントローラ内のレジスタ(MODE)で行なわれる。

グループで共通なレジスタやフラグは共有バスを通して同時に書き替えることができる。つまり，スケジューラが共有バスにグループを特定した同期コントローラへの命令を出力でき，それによってグループ内のコントローラのレジスタが書き替わる。同様にしてスケジューラは自分が動作しているプロセッサ以外に接続している特定のコントローラのレジスタも変更できる。このようにスケジューラによって設定されるレジスタ類は図6 では GR, MODE, MASK, PRQF, MPRQF, MNWRF, TVPS である。この内 MASK は割り込みマスクで，これがセットされるとコントローラからプロセッサへの割り込みが禁止される。PRQF, MPRQF, MNWRF, TVPS はプロセッサ割当てに関する情報を格納するレジスタ類である。PRQF, MPRQF, MNWRF はそれぞれ 2.3.節の #PQR, #MPRQ, #MNWR に対応するフラグで，計数が 0 であればフラグはリセット，計数が 0 以外ではフラグがセットされる。TVPS は条件(3)〜(7) で #MWC の数と比較すべき数を設定するためのレジスタである。例えば，条件(3)(4)の検出にはこれを #MGC に設定する。

まず，モード1 の動作を述べる。同期コントローラ内には前述のようにプロセッサ割当てに関する情報を格納するレジスタ類があり，スケジューラにより値がセットしてある。#MWC については同期コントローラが同期信号バスを監視して常時把握している。システムの状態について(3)〜(7)の条件のチェックをハードウェアで行ない，条件を満たすとプロセッサに割り込み信号を出力し，スケジューラによるプロセスの再スケジューリングを要求する。割り込みが集中しないようにグループ内では同時に一つの割り込みしか発生しないように同期コントローラに優先順位を付けておく。

次に，モード2 の動作について述べる。同期コントローラ内に前出のレジスタやフラグの他に読出し専用のバリア同期成立フラグ　(BF)　と内部状態記憶のためのフラグ(BSOF) が存在する。但し，スケジューラからは副作用なしに読み書きが可能である。BF は　MNWRF がリセッ

ト（'0'）で，且つ自分の属するグループの総てのプロセッサが同期待ち状態のときに '1'にセットされる。この時，副作用として自分の同期信号線つまり SOF を'0'にリセットする。そして，この BF が値'1'として BSOF が'1'である期間にプロセッサから読み出されると，まず，BSOF を'0'にして，MPRQF が'0'でない場合コントローラ内の MNWRF を'1'にセットし，この後，BF をリセットする。以上の動作をコントローラが自動的に行う。BSOF が'0'である場合は BF のリセットのみを行う。プログラムはこの BF を使って待ち合わせを行う。また，モード1では SOF を明示的にセットして同期信号線を'1'にしているが，このモードでは SOF の自動リセット同様に BF による同期成立後最初の読出しで自動的にセットできるのでビジー・ウェイト処理内でセットする必要はない。BSOF は SOF のセットと同時にセットされる。結局，待ち合わせの処理は図7 のフローチャートのようになる。スケジューラによるプロセスの生成削除の際の誤動作を防ぐため以下のように制御する。例えば，バリア同期に参加するプロセスの生成時はコントローラ内の MNWRF に'1'を設定しておき，割り込みは禁止しておく，そしてプロセスが総て生成された後に正しい MNWRF を設定し，割り込みを許可する。また，同期コントローラは MNWRF が'1'で内部的に SOF がセットされると，プロセスを切り換えさせるためプロセッサに割り込みを発生させる。この時，切り替えたプロセスをバリア同期成立後にプロセッサに再割り付けする際は，SOF と BSOF を'0'に BF を'1'に設定する。

同期コントローラ内のレジスタはプロセス切り換え発生毎に適宜更新され，プロセスが入替えられるプロセッサの同期コントローラの SOF, BSOF, BF, MNWRF 等は入替え前の値が退避され，再びそのプロセスが割り付けられた際に，再設定される。

4. 階層化されたスケジューラ

プロセッサの割当て待ちのプロセスが複数ある場合，どのプロセスをプロセス切り換え時にプロセッサに割り付けるかが効率上問題である。また，プロセッサ台数以上のプロセスで並列処理を行う場合は必ず再スケジューリングが起こり，プロセスの粒度が細かいときにはその頻度が非常に大きい。そこで，そのような並列処理にも対応できるように再スケジューリングのオーバーヘッドを極力削減する必要がある。

通常スケジューリングは OS のカーネル内で行なわれる。よって，ユーザーのプロセスが再スケジューリングを行ってもらうためにはカーネルへのシステム呼出しを行う必要がある。しかし，カーネル内のスケジューラではユーザーのアプリケーション毎のきめ細かなスケジューリングは不可能である。また，OS のカーネルとユーザーのプロセスが多くのデータを共有するのはそれを通知しあう手順が複雑になり，オーバーヘッドも増大する。さらに，システム呼出しそれ自身のオーバーヘッドがかなり重いので，軽くプロセスの切り換えを行うのは難しい[8]。そこで，スケジューラを階層化して，従来の実プロセッサを割り付けるためのスケジューラをカーネル・スケジューラと呼び，その制御下にユーザー・スケジューラを設ける（ 図8参照 ）。ユーザー・スケジューラは同一のアプリケーションを協調して処理するプロセス・グループ毎に設けられ，スケジューリング方式はユーザーが処理に応じて決定できる。カーネル・スケジューラは実プロセッサ全体の管理を行ない，プロセス・グループ毎にまとめて実プロセッサを割り付ける。ユーザー・スケジューラはグループに割当てられたプロセッサ内でのスケジューリングを実行する。また，ユーザー・スケジューラはカーネル側（カーネル・モード）ではなくユーザー側（ユーザー・モード）で実行される。そこで，プロセス切り換えの際，システム呼出し等のオーバーヘッドがなく軽い。

プロセス切り換えの条件が成立した場合，ユーザー・スケジューラに制御を渡し，そのプロセス・グループに適したスケジューリング・アルゴリズムで次に割当てるべきプロセスを選び，そのプロセスに制御を渡す。これにより，システム呼出しのオーバーヘッドなしにプロセスをグループ内で切り換える。

カーネル・スケジューラは各プロセス・グループからプロセッサの要求台数を受け取り，なるべく要求を満たすようにスケジューリングを行う。カーネル・スケジューラはタイム・シェアリングのためのタイマ割り込み時または周辺装置の入出力を伴うシステム呼出し時またはユーザー・スケジューラからのプロセッサ資源の追加要求・返還時に起動する。カーネルからユーザーのプロセスに制御が移るときはグループのユーザー・スケジューラを通して制御が移る。カーネル・スケジューラによるスケジューリングの

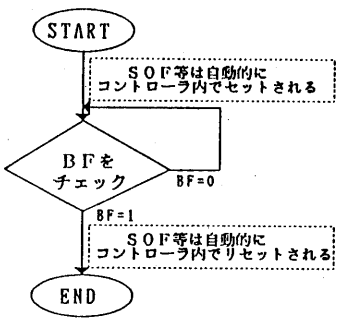

図7 BFを使った軽いバリア同期処理

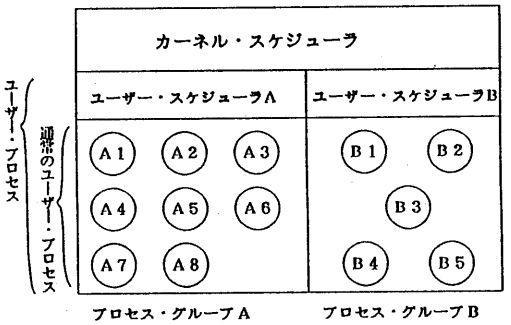

図8 階層化されたスケジューラ

結果，空きプロセッサが必要になった場合は優先度の低いプロセス・グループのプロセッサを取り上げる（プリエンプトする）。プリエンプトされたプロセスのプロセッサ・コンテクストはユーザー・スケジューラからアクセス可能なメモリに格納する。

5. ユーザーに解放された割り込み機構

同期におけるプロセス切り換えの条件判断をハードウェア化して，割り込みで条件成立をプロセッサに通知する場合，この割り込みでユーザー・スケジューラが起動するようにしておく。実行モードにカーネル側とユーザー側の区別があるプロセッサでは割り込み後はカーネル側に制御が切り替わる。このような従来型のプロセッサをシステムで使う時は，この割り込み時にはなるべくオーバーヘッドなしにユーザー側に制御を返す必要がある。この点に関し，積極的に以下のように割り込み機能を階層化したプロセッサを用意すれば，オーバーヘッドの少ないユーザー・モード内での割り込みが行える。優先度の違う外部割り込みをプロセッサに用意し，優先度の低い外部割り込みの内のいくつかにユーザー・モードの割り込みを設ける。つまり，割り込みが発生したらユーザーが設定したルーチンにユーザー・モード内で制御を移す。残りの外部割り込みは従来通りカーネル・モードへの割り込みである。また，ユーザー側ではカーネル・モードの割り込みを禁止できないが，ユーザー・モードの割り込みは割り込みのマスクの切り換え（割り込み許可・不許可の切り換え）もユーザー側で自由にできる。このユーザー・モードの割り込みを同期機構からの割り込みや同じグループ内のプロセッサへの非同期の通信に使うと，カーネルに制御が移行しないのでオーバーヘッドを減らすことができる。同期機構からの割り込みでは直接ユーザー・スケジューラに制御が移るようにしておく。

6.おわりに

共有メモリ型のマルチプロセッサシステムにおいて，プロセス間の同期に伴うオーバーヘッドと無意味なスケジューリングを極力抑えて高性能な並列処理を可能にするために以下の方式並びに機構を提案した。オーバーヘッドを減らすため共有メモリを介しての同期を採用し，システムのプロセッサ資源のプロセスへの割当に関する情報をユーザーからアクセス可能にしておき，ビジー・ウェイトの同期待ちのループで同期変数のチェックだけではなく，プロセッサ割当てに関する情報もチェックする新しいビジー・ウェイト同期方式。そのチェックの結果，同期が暫く成立しない状況と判断されるときはビジー・ウェイト中のプロセスは自らプロセスの実行を中断し，スケジューラに制御を移し，再スケジューリングを行わせプロセッサへのプロセスの割当てを変更する。また，プロセッサ割当てに関する情報をチェックすることで生ずるオーバーヘッドを軽くするために支援ハードウェアを設け，再スケジューリングの開始は割り込みで通知されるようにする同期機構。スケジューリングの際に最適なスケジューリングが行えるようにスケジューラのユーザーによるカスタマイズを OSの管理下で可能にする，ユーザーのスケジューラとカーネルのスケジューラに階層化されたスケジューラ。さらに，再スケジューリングを指示する割り込みのオーバーヘッドを減らすためとユーザーのスケジューラが複数のプロセッサに対してなるべく小さなオーバーヘッドで非同期通信できるために，ユーザーに解放された割り込みを持ったプロセッサの割り込み機構。

これらの方式と機構の効果の大きさは，実行されるプログラムの性質やシステムの負荷状況によって大きく変化すると考えられる。今後の課題としては何らかの仮定を設けてその効果の大きさを評価することである。

また，階層化されたスケジューラにおける，カーネル・スケジューラによるプリエンプションの方式については別途詳細に報告する予定である。

なお，第39回情報処理学会全国大会で NTTソフトウェア研究所の高橋直久氏が本稿のビジー・ウェイト同期方式を相互排除へ利用した場合と同じ原理の機構を提案された(9)。

謝辞

いつもご討論いただいている同僚の森山孝男氏と渦原茂氏に感謝いたします。特に，階層化されたスケジューラは彼らとの議論が基本アイデアとなっています。並びに，本稿の作成に協力と助言をいただいた田中朋之氏，本研究の機会を与えていただいた鈴木則久所長に深謝します。

文献

1. 松本: 細粒度並列実行支援マルチプロセッサの検討. 信学技報 Vol.89 No.167, CPSY 89-37 (August 1989) pp. 37-42.
2. 松本: 細粒度並列実行支援機構. 情処研報 Vol.89 No.60, 89-ARC-77-12 (July 1989) pp. 91-98.
3. 松本: 一般化されたバリア型同期機構. 情処第39回全大, 5X-9 (October 1989) pp. 1880-1881.
4. M.Papamarcos and J.Patel: A low-overhead coherence solution for multiprocessors with private cache memories. *Proc. 11th Annual IEEE Int. Symp. on Computer Architecture* (1984) pp. 348-354.
5. T.J.Teixeira and R.F.Gurwitz: Stellix:UNIX for a Graphics Supercomputer. *Proc. of the Summer 1988 USENIX Conf.* (June 1988) pp. 321-330.
6. B.Beck et al.: VLSI Assist for a Multiprocessor. *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (October 1987) pp. 10-20.
7. C.D.Polychronopoulos: Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design. *IEEE Trans. Comput., vol.37, no.8* (August 1988) pp. 991-1004.
8. 渦原、森山: マルチプロセッサシステムにおける ライトウエイトプロセス機構. ソフトウェア科学会第6回大会, C6-3 (October 1989) pp. 353-356.
9. 高橋: メモリ共有型マルチプロセッサにおける 小粒度プロセスの相互排除機構. 情処第39回全大, 4P-2 (October 1989) pp. 1213-1214.